



12 基础篇 | TCP收发包过程会受哪些配置项影响？

2020-09-15 邵亚方

Linux内核技术实战课

[进入课程 >](#)



讲述：邵亚方

时长 16:14 大小 14.88M



你好，我是邵亚方。我们这节课来讲一下，TCP 数据在传输过程中会受到哪些因素干扰。

TCP 收包和发包的过程也是容易引起问题的地方。收包是指数据到达网卡再到被应用程序开始处理的过程。发包则是应用程序调用发包函数到数据包从网卡发出的过程。你应该对 TCP 收包和发包过程中容易引发的一些问题不会陌生，比如说：

网卡中断太多，占用太多 CPU，导致业务频繁被打断；

应用程序调用 `write()` 或者 `send()` 发包，怎么会发不出去呢；

数据包明明已经被网卡收到了，可是应用程序为什么没收到呢；

我想要调整缓冲区的大小，可是为什么不生效呢；



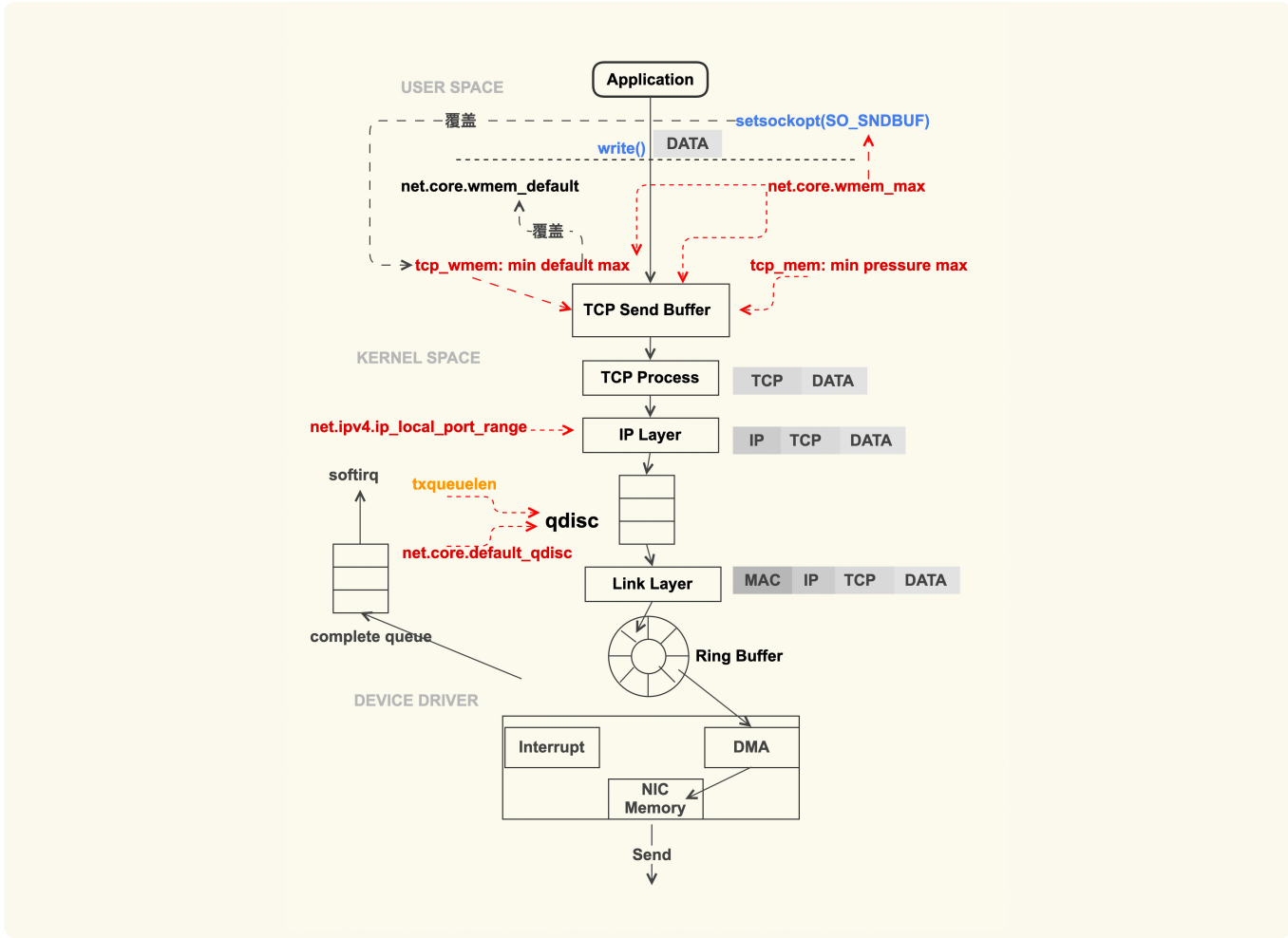
是不是内核缓冲区满了从而引起丢包，我该怎么观察呢；

...

想要解决这些问题呢，你就需要去了解 TCP 的收发包过程容易受到哪些因素的影响。这个过程中涉及到很多的配置项，很多问题都是这些配置项跟业务场景不匹配导致的。

我们先来看下数据包的发送过程，这个过程会受到哪些配置项的影响呢？

TCP 数据包的发送过程会受什么影响？



TCP数据包发送过程

上图就是一个简略的 TCP 数据包的发送过程。应用程序调用 `write(2)` 或者 `send(2)` 系列系统调用开始往外发包时，这些系统调用会把数据包从用户缓冲区拷贝到 TCP 发送缓冲区（TCP Send Buffer），这个 TCP 发送缓冲区的大小是受限制的，这里也是容易引起问题的地方。

TCP 发送缓冲区的大小默认是受 `net.ipv4.tcp_wmem` 来控制：

```
net.ipv4.tcp_wmem = 8192 65536 16777216
```

tcp_wmem 中这三个数字的含义分别为 min、default、max。TCP 发送缓冲区的大小会在 min 和 max 之间动态调整，初始的大小是 default，这个动态调整的过程是由内核自动来做的，应用程序无法干预。自动调整的目的，是为了在尽可能少的浪费内存的情况下来满足发包的需要。

tcp_wmem 中的 max 不能超过 net.core.wmem_max 这个配置项的值，如果超过了，TCP 发送缓冲区最大就是 net.core.wmem_max。通常情况下，我们需要设置 net.core.wmem_max 的值大于等于 net.ipv4.tcp_wmem 的 max：

```
net.core.wmem_max = 16777216
```

对于 TCP 发送缓冲区的大小，我们需要根据服务器的负载能力来灵活调整。通常情况下我们需要调大它们的默认值，我上面列出的 tcp_wmem 的 min、default、max 这几组数值就是调大后的值，也是我们在生产环境中配置的值。

我之所以将这几个值给调大，是因为我们在生产环境中遇到过 TCP 发送缓冲区太小，导致业务延迟很大的问题，这类问题也是可以使用 systemtap 之类的工具在内核里面打点来进行观察的（观察 sk_stream_wait_memory 这个事件）：

[复制代码](#)

```
1 # sndbuf_overflow.stp
2 # Usage :
3 # $ stap sndbuf_overflow.stp
4 probe kernel.function("sk_stream_wait_memory")
5 {
6     printf("%d %s TCP send buffer overflow\n",
7           pid(), execname())
8 }
```

如果你可以观察到 sk_stream_wait_memory 这个事件，就意味着 TCP 发送缓冲区太小了，你需要继续去调大 wmem_max 和 tcp_wmem:max 的值了。

应用程序有的时候会很明确地知道自己发送多大的数据，需要多大的 TCP 发送缓冲区，这个时候就可以通过 setsockopt(2) 里的 SO_SNDBUF 来设置固定的缓冲区大小。一旦进行

了这种设置后，`tcp_wmem` 就会失效，而且这个缓冲区大小设置的是固定值，内核也不会对它进行动态调整。

但是，`SO_SNDBUF` 设置的最大值不能超过 `net.core.wmem_max`，如果超过了该值，内核会把它强制设置为 `net.core.wmem_max`。所以，如果你想要设置 `SO_SNDBUF`，一定要确认好 `net.core.wmem_max` 是否满足需求，否则你的设置可能发挥不了作用。通常情况下，我们都不会通过 `SO_SNDBUF` 来设置 TCP 发送缓冲区的大小，而是使用内核设置的 `tcp_wmem`，因为如果 `SO_SNDBUF` 设置得太大就会浪费内存，设置得太小又会引起缓冲区不足的问题。

另外，如果你关注过 Linux 的最新技术动态，你一定听说过 eBPF。你也可以通过 eBPF 来设置 `SO_SNDBUF` 和 `SO_RCVBUF`，进而分别设置 TCP 发送缓冲区和 TCP 接收缓冲区的大小。同样地，使用 eBPF 来设置这两个缓冲区时，也不能超过 `wmem_max` 和 `rmem_max`。不过 eBPF 在一开始增加设置缓冲区大小的特性时并未考虑过最大值的限制，我在使用的过程中发现这里存在问题，就给社区提交了一个 PATCH 把它给修复了。你感兴趣的话可以看下这个链接：[🔗 bpf: sock recvbuff must be limited by rmem_max in bpf_setsockopt\(\)](#)。

`tcp_wmem` 以及 `wmem_max` 的大小设置都是针对单个 TCP 连接的，这两个值的单位都是 Byte（字节）。系统中可能会存在非常多的 TCP 连接，如果 TCP 连接太多，就可能导致内存耗尽。因此，所有 TCP 连接消耗的总内存也有限制：

```
net.ipv4.tcp_mem = 8388608 12582912 16777216
```

我们通常也会把这个配置项给调大。与前两个选项不同的是，该选项中这些值的单位是 Page（页数），也就是 4K。它也有 3 个值：min、pressure、max。当所有 TCP 连接消耗的内存总和达到 max 后，也会因达到限制而无法再往外发包。

因 `tcp_mem` 达到限制而无法发包或者产生抖动的问题，我们也是可以观测到的。为了方便地观测这类问题，Linux 内核里面预置了静态观测点：`sock_exceed_buf_limit`。不过，这个观测点一开始只是用来观察 TCP 接收时遇到的缓冲区不足的问题，不能观察 TCP 发送时遇到的缓冲区不足的问题。后来，我提交了一个 patch 做了改进，使得它也可以用来观察 TCP 发送时缓冲区不足的问题：[🔗 net: expose sk wmem in](#)

`sock_exceed_buf_limit tracepoint`，观察时你只需要打开 `tracepoint`（需要 4.16+ 的内核版本）：

```
$ echo 1 >
/sys/kernel/debug/tracing/events/sock/sock_exceed_buf_limit/enable
```

然后去看是否有该事件发生：

```
$ cat /sys/kernel/debug/tracing/trace_pipe
```

如果有日志输出（即发生了该事件），就意味着你需要调大 `tcp_mem` 了，或者是需要断开一些 TCP 连接了。

TCP 层处理完数据包后，就继续往下来到了 IP 层。IP 层这里容易触发问题的地方是 `net.ipv4.ip_local_port_range` 这个配置选项，它是指和其他服务器建立 IP 连接时本地端口（local port）的范围。我们在生产环境中就遇到过默认的端口范围太小，以致于无法创建新连接的问题。所以通常情况下，我们都会扩大默认的端口范围：

```
net.ipv4.ip_local_port_range = 1024 65535
```

为了能够对 TCP/IP 数据流进行流控，Linux 内核在 IP 层实现了 `qdisc`（排队规则）。我们平时用到的 TC 就是基于 `qdisc` 的流控工具。`qdisc` 的队列长度是我们用 `ifconfig` 看到的 `txqueuelen`，我们在生产环境中也遇到过因为 `txqueuelen` 太小导致数据包被丢弃的情况，这类问题可以通过下面这个命令来观察：

```
$ ip -s -s link ls dev eth0
...
TX: bytes packets errors dropped carrier collsns
3263284 25060 0 0 0 0
```

如果观察到 `dropped` 这一项不为 0，那就有可能是 `txqueuelen` 太小导致的。当遇到这种情况时，你就需要增大该值了，比如增加 `eth0` 这个网络接口的 `txqueuelen`：

```
$ ifconfig eth0 txqueuelen 2000
```

或者使用 ip 这个工具：

```
$ ip link set eth0 txqueuelen 2000
```

在调整了 txqueuelen 的值后，你需要持续观察是否可以缓解丢包的问题，这也便于你将它调整到一个合适的值。

Linux 系统默认的 qdisc 为 pfifo_fast（先进先出），通常情况下我们无需调整它。如果你想使用  TCP BBR 来改善 TCP 拥塞控制的话，那就需要将它调整为 fq（fair queue, 公平队列）：

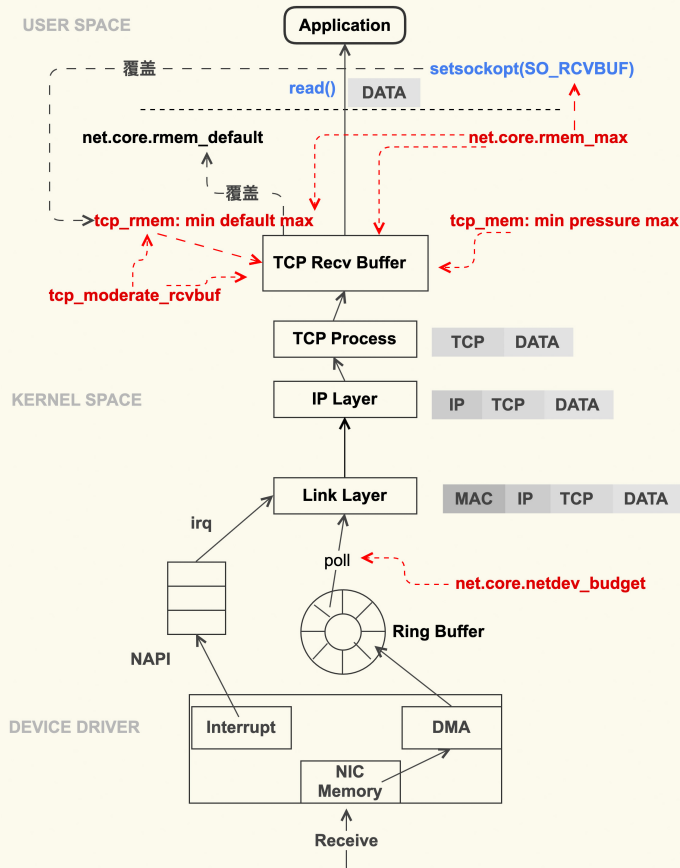
```
net.core.default_qdisc = fq
```

经过 IP 层后，数据包再往下就会进入到网卡了，然后通过网卡发送出去。至此，你需要发送出去的数据就走完了 TCP/IP 协议栈，然后正常地发送给对端了。

接下来，我们来看下数据包是怎样收上来的，以及在接收的过程中会受哪些配置项的影响。

TCP 数据包的接收过程会受什么影响？

TCP 数据包的接收过程，同样也可以用一张图来简单表示：



TCP数据包接收过程

从上图可以看出，TCP 数据包的接收流程在整体上与发送流程类似，只是方向是相反的。数据包到达网卡后，就会触发中断（IRQ）来告诉 CPU 读取这个数据包。但是在高性能网络场景下，数据包的数量会非常大，如果每来一个数据包都要产生一个中断，那 CPU 的处理效率就会大打折扣，所以就产生了 NAPI（New API）这种机制让 CPU 一次性地去轮询（poll）多个数据包，以批量处理的方式来提升效率，降低网卡中断带来的性能开销。

那在 poll 的过程中，一次可以 poll 多少个呢？这个 poll 的个数可以通过 sysctl 选项来控制：

```
net.core.netdev_budget = 600
```

该控制选项的默认值是 300，在网络吞吐量较大的场景中，我们可以适当地增大该值，比如增大到 600。增大该值可以一次性地处理更多的数据包。但是这种调整也是有缺陷的，因为这会导致 CPU 在这里 poll 的时间增加，如果系统中运行的任务很多的话，其他任务的调度延迟就会增加。

接下来继续看 TCP 数据包的接收过程。我们刚才提到，数据包到达网卡后会触发 CPU 去 poll 数据包，这些 poll 的数据包紧接着就会到达 IP 层去处理，然后再达到 TCP 层，这时就会面对另外一个很容易引发问题的地方了：TCP Receive Buffer（TCP 接收缓冲区）。

与 TCP 发送缓冲区类似，TCP 接收缓冲区的大小也是受控制的。通常情况下，默认都是使用 `tcp_rmem` 来控制缓冲区的大小。同样地，我们也会适当地增大这几个值的默认值，来获取更好的网络性能，调整为如下数值：

```
net.ipv4.tcp_rmem = 8192 87380 16777216
```

它也有 3 个字段：min、default、max。TCP 接收缓冲区大小也是在 min 和 max 之间动态调整，不过跟发送缓冲区不同的是，这个动态调整是可以通过控制选项来关闭的，这个选项是 `tcp_moderate_rcvbuf`。通常我们都是打开它，这也是它的默认值：

```
net.ipv4.tcp_moderate_rcvbuf = 1
```

之所以接收缓冲区有选项可以控制自动调节，而发送缓冲区没有，那是因为 TCP 接收缓冲区会直接影响 TCP 拥塞控制，进而影响到对端的发包，所以使用该控制选项可以更加灵活地控制对端的发包行为。

除了 `tcp_moderate_rcvbuf` 可以控制 TCP 接收缓冲区的动态调节外，也可以通过 `setsockopt()` 中的配置选项 `SO_RCVBUF` 来控制，这与 TCP 发送缓冲区是类似的。如果应用程序设置了 `SO_RCVBUF` 这个标记，那么 TCP 接收缓冲区的动态调整就是关闭，即使 `tcp_moderate_rcvbuf` 为 1，接收缓冲区的大小始终就为设置的 `SO_RCVBUF` 这个值。

也就是说，只有在 `tcp_moderate_rcvbuf` 为 1，并且应用程序没有通过 `SO_RCVBUF` 来配置缓冲区大小的情况下，TCP 接收缓冲区才会动态调节。

同样地，与 TCP 发送缓冲区类似，`SO_RCVBUF` 设置的值最大也不能超过 `net.core.rmem_max`。通常情况下，我们也需要设置 `net.core.rmem_max` 的值大于等于 `net.ipv4.tcp_rmem` 的 max：

```
net.core.rmem_max = 16777216
```

我们在生产环境中也遇到过，因达到了 TCP 接收缓冲区的限制而引发的丢包问题。但是这类问题不是那么好追踪的，没有一种很直观地追踪这种行为的方式，所以我便在我们的内核里添加了针对这种行为的统计。

为了让使用 Linux 内核的人都能很好地观察这个行为，我也把我们的实践贡献给了 Linux 内核社区，具体可以看这个 commit: [tcp: add new SNMP counter for drops when try to queue in rcv queue](#)。使用这个 SNMP 计数，我们就可以很方便地通过 netstat 查看，系统中是否存在因为 TCP 接收缓冲区不足而引发的丢包。

不过，该方法还是存在一些局限：如果我们想要查看是哪个 TCP 连接在丢包，那么这种方式就不行了，这个时候我们就需要去借助其他一些更专业的 trace 工具，比如 eBPF 来达到我们的目的。

课堂总结

好了，这节课就讲到这里，我们简单回顾一下。TCP/IP 是一个很复杂的协议栈，它的数据包收发过程也是很复杂的，我们这节课只是重点围绕这个过程中最容易引发问题的地方来讲述的。我们刚才提到的那些配置选项都很容易在生产环境中引发问题，并且也是我们针对高性能网络进行调优时必须要去考虑的。我把这些配置项也总结为了一个表格，方便你来查看：

配置项	说明
net.ipv4.tcp_wmem	TCP发送缓冲区大小，分别有min、default、max这三个值，TCP发送缓冲区大小会在min和max之间动态调整。如果通过SO_RCVBUF来设置TCP发送缓冲区的大小，那么发送缓冲区就不会自动调整大小了。
net.core.wmem_max	TCP发送缓冲区的最大值，通常我们都需要调高该值来获得更好的网络性能。
net.ipv4.tcp_mem	系统中TCP连接最多可以消耗的内存，接收缓冲区和发送缓冲区都受它控制，通常我们也需要调大该值。
net.ipv4.ip_local_port_range	本地端口范围，如果该值设置得太小，就可能导致无法创建新的连接。
txqueuelen	qdisc的大小，可以通过ifconfig或者ip命令来调整该值，通常情况下也是需要调大该值。
net.core.default_qdisc	默认qdisc，通常情况下我们无需调整该值，但是有些场景下我们必须要去调整该值，比如说如果TCP拥塞控制使用的是BBR，那就需要将它配置为fq这种方式。
net.core.netdev_budget	接收到网卡中断后，CPU从ring buffer中一次最多可以读取多少个数据包。通常情况下我们需要调大该值来获取更好的网络性能。
net.ipv4.tcp_rmem	TCP接收缓冲区大小，分别有min、default、max这三个值，TCP接收缓冲区大小会在min和max之间动态调整。如果关闭了tcp_moderate_rcvbuf或者通过SO_RCVBUF来设置TCP接收缓冲区大小，那么接收缓冲区就不会自动调整大小了。
net.ipv4.tcp_moderate_rcvbuf	是否允许动态调整TCP接收缓冲区的大小，该配置项存在的意义是可以影响TCP拥塞控制，进而再影响发送方的行为。
net.core.rmem_max	TCP接收缓冲区的最大值，通常我们都需要调高该值来获得更好的网络性能。

这些值都需要根据你的业务场景来做灵活的调整，当你不知道针对你的业务该如何调整时，你最好去咨询更加专业的人员，或者一边调整一边观察系统以及业务行为的变化。

课后作业

我们这节课中有两张图，分别是 TCP 数据包的发送过程 和 TCP 数据包的接收过程，我们可以看到，在 TCP 发送过程中使用到了 qdisc，但是在接收过程中没有使用它，请问是因为什么？我们可以在接收过程中也使用 qdisc 吗？欢迎你在留言区与我讨论。

感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友，我们下一讲见。

提建议

更多课程推荐

数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



立省 ¥40

破 90000 订阅特惠，到手价 ¥89

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 基础篇 | TCP连接的建立和断开受哪些系统配置影响？

下一篇 13 案例篇 | TCP拥塞控制是如何导致业务性能抖动的？

精选留言 (5)

写留言



邵亚方 置顶

2020-10-11

课后作业答案：

- 在 TCP 发送过程中使用到了 qdisc，但是在接收过程中没有使用它，请问是为什么？我们可以在接收过程中也使用 qdisc 吗？

qdisc的主要目的是流控，流控一般都是在发送端进行控制；对于接收端而言，它已经收到这个包了，再进行流控的话，也就只有选择性的丢包。如果在接收端也使用qdisc之类的...

展开 ∨



我来也

2020-09-15

终于到TCP篇了.

看了文中的`TCP数据包发送过程`图,有几个疑问:

1. 如果用tcpdump抓包,它是在哪一层抓的包呢?(IP Layer / Link Layer)

最近遇到的问题,就是调用write函数有返回值.但是tcpdump抓包来看,并没有迹象....

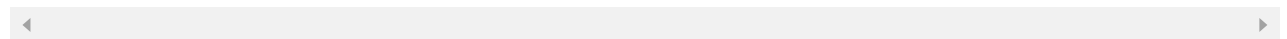
展开 ∨

作者回复: 1. 这个tcpdump的原理,我们在后面的课程里会讲到,是在link layer来抓包的。

2. 对的, 会按需调整, 调整后会影响之前的连接, 因为在检查缓冲区大小时会用到这些全局变量。

3. 对的 跟是否设置了阻塞模式有关。

这个blog分享得不错, 赞!



💬 1

👍 2



王崧霁

2020-09-19

流控应该在上游发送端控制, 接收端有个开关net.core.devbudget也是控制发端行为

💬

👍 1



solar

2020-09-28

用户通过openvpn连接服务器, 然后使用的udp报文, 然后测速发现连接VPN后的速度比连接VPN前要慢的多, 但是服务器的入口带宽其实还剩余很多, 这个是由于udp的缓存设置导致的吗? 老师能解答下吗?

💬 1

👍



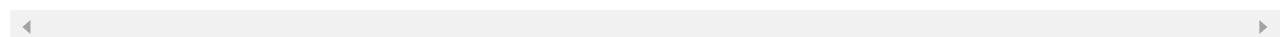
feihui

2020-09-20

根据老师文中对 qdisc 长度的设置, 这个队列应该是网卡独享的设备, 此外使用 bbr 需要更改改队列行为, 猜测一部分是跟发送优先级有关, 而这在接收端没必要

展开 ∨

作者回复: 对的 qdisc可以做流控 这主要是在发送端才需要做流控 接收端只能通过拥塞控制来影响发送端的行为。



💬

👍

